# Experience: Ewok: Lessons in Unreliability of Wear OS Ecosystem through State-Aware Fuzzing


## ABSTRACT

As smartwatches and health monitoring devices running Wear OS (formerly Android Wear) gain popularity, it is important to evaluate the resilience of the operating system and its application ecosystem. Prior work has analyzed the OS in isolation without considering the state of the device, hardware sensors, or the interaction between the wearable and the mobile device. In this paper, we study the impact of the state of a wearable device on the overall reliability of the applications running in Wear OS. We use characteristics of wearable apps, such as use of sensors and mobile-wearable pairing, to derive a state model and use this model to target specific fuzz injection campaigns against a set of popular wearable apps. Our experiments revealed an abundance of improper exception handling, device reboot due to excessive sensor use, presence of legacy codes, and error propagation across mobile and wearable devices. We present some lessons for developers to improve the reliability of the wearable ecosystem.


## 1. INTRODUCTION

Wear OS is one of the dominant OS for wearable devices and three major versions have been released by its developer, Google, since 2014. The OS is based on Android, and both share similar features (e.g. kernel, programming model, apps life cycle, development framework etc.). However, there are major differences between Android and Wear OS applications. While Android applications are mostly self-contained, Wear applications often need to delegate tasks to a mobile counterpart. This is primarily driven by the limited display area on the watch and the difficulty of executing interactive work (such as typing) on the wearable[1]. Limited user interaction leads to development of applications that have more service components and fewer GUI components [26]. To compensate for the lack of input methods such as a reasonable-sized software keyboard, recent generation of wear OS applications are heavily reliant on Google Assistant for voice commands and user inputs. Moreover, wearable devices are often fitted with sensors that are not available on smartphones (e.g., heart rate monitor, pulse oximeter). Altogether, wearable applications are generally more dependent on device sensors, have complex communication patterns (both intra-device and inter-device), and are context aware.

Although there have been several previous works focusing on the reliability of the Android ecosystem [11, 20,21,29,30], there are only few that study the wearable ecosystem [9, 27]. However, neither of these consider inter-device communication (between mobile and wearable) in wearable apps and their dependence on sensors. In [9], the authors presented a black-box fuzzing tool named QGJ which tests the robustness of wearable apps based on a set of fuzz campaigns. QGJ is agnostic about the state of a wearable application (in terms of sensor activity) and does not fuzz inter-device communication. We argue that fuzzing would be more effective in detecting bugs if we consider the state of an application and apply different fuzzing strategies based on the state. In this paper, we present Ewok[2], a state-aware fuzzing tool for wearable devices. Ewok does not need source code of the wearable applications and does not need any modification to the wearable OS. We then proceed to do a detailed study of 15 popular apps from the Google Play Store in the categories of health & fitness, maps and navigation, shopping and personalization. Using our new stateful fuzzer for Wear OS, we inject over 1M Intents between inter-device and intra-device interactions. We uncover some foundational reliability weak spots in the design of Wear OS and several vulnerabilities that are common across multiple apps. Based on the experience from these injection campaigns, we provide specific recommendations to improve the reliability of software on wearable devices.

Our results reveal several interesting failure cases– abundance of improper exception handling, presence of legacy codes in Wear OS, and error propagation across

[1]We will often use smartwatch as a stand-in for a wearable, as it is the most popular wearable platform, though there are many other kinds of wearables too such as smart glasses. Wear OS is a dominant platform for smartwatches.

[2]Ewok is named after the tiny furry warriors of the planet Endor of Star Wars galaxy who were instrumental in the destruction of the Death Star II.

devices. We also found that under certain experimental setup (i.e., states) we could predictably crash the Android runtime by sending fuzz inputs from user-level apps. We recommend stronger definition of message types and more proactive use of IDEs in catching exception handling bugs at development time. Overall, we found that our state-aware injections are more effective in triggering crashes and reboots than state-agnostic fuzzers such as QGJ.

Our key contributions in this paper are:

**1. State-aware fuzzing:** We present a state-aware fuzzing tool for wearable apps. Our tool, Ewok, can infer the state-model of a wearable application without source code access, guide it to specific states, and then run targeted fuzz campaigns. Our tool can be downloaded from [URL hidden for anonymity].

**2. Higher failure activation:** We show that the stateful fuzzing can increase the fault activation rate on Wear OS compared to a state-agnostic approach and compare to three prior tools (QGJ, Monkey, Droidbot). We found several interesting failure cases through these experiments and proposed solutions to make the wearable ecosystem more robust.

**3. Catastrophic failures:** We demonstrate that it is possible to trigger catastrophic failures on Wear OS from user space, without privileges. Specifically, we identify that Wear OS is vulnerable in certain states of the device, namely, where there is high sensor activity coupled with synchronization between mobile and wearable devices.

## 2. BACKGROUND

Wear OS by Google is an operating system for wearable devices based on Android. Compared to their mobile counterpart, Wear OS apps have a minimalist UI design, more focus on services (which are background processes), and have a sensor-rich application context. Although Wear OS supports standalone applications, a significant majority of wearable apps are tightly coupled with a mobile counterpart. As a result, most of the wearable applications frequently communicate with their companion apps in the mobile to share data or to exchange IPC messages. Communication between wearable applications can happen either intra-device or inter-device. Intra-device communication typically happens through the binder IPC framework [11] and takes the form of Intent messages. For inter-device communication, several types of messages may be exchanged:

*Message passing and data synchronization.* This is commonly used to share data or to enable the IPC communication among devices.

*Delegation of actions from the wearable to the mobile.* The operation starts in the wearable, continues on the phone, and returns to the wearable once completed.

*Bridged Notifications.* These notifications allow to communicate brief messages to the user, which sometimes can also be used to control the flow of the application.

In this paper, we evaluate the robustness of wearable applications by fuzzing both intra-device messages (Intents) and inter-device messages (the first two kinds).

## 3. DESIGN

A key motivation behind Ewok comes from the fact that behaviors of wearable applications are often state (or context) dependent, where state can either be a state of the application (e.g., operation being performed) or state of the device (e.g., physical location or movement of the wearable device). We hypothesize that this translates to failures of the wearable software stack that are also state-dependent. To validate this hypothesis, we design and implement Ewok with the following objectives:

1) to define a state model that is widely applicable to most wearable apps,
2) to fuzz popular apps using this state model, and
3) to show that the proposed tool can activate more faults compared to a state-agnostic fuzzer.

In the subsequent sections, we elaborate how each of these objectives is met.

### 3.1 State Model

During our initial fuzz experiments with wearable applications, we found that several applications crashed while accessing sensors. This is consistent with the observations found in QGJ [9]. Moreover, sensors often determine the *context* of a wearable application (e.g., location or accelerometer readings). We therefore decided to use sensor activation as a characteristic of the wearable state. Another unique characteristic of wearable applications is that these are often tightly coupled with their mobile counterpart applications. This means that a wearable app needs a mobile app to function. After various sensor data are collected at the wearable, this data is sent to the mobile, or control commands are sent from the mobile app to the wearable app. This background *data synchronization* may lead to new bugs not seen in mobile apps. Therefore, in our model, a combination of sensor activation and data synchronization activity together determine the state of the application.

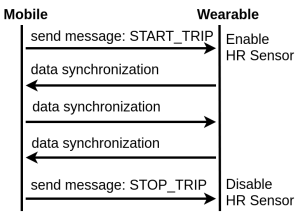


**Figure 1: Inter-device communication for a fitness app.**

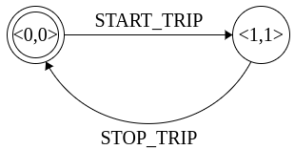


**Figure 2: Example state model for a fitness app.**

As an example, consider a fitness application that helps people keep track of their workout. These ap-

plications commonly include two paired apps, installed on the mobile and the wearable device. Before starting the workout, a person uses the mobile to indicate the app that she is going to start (`START_TRIP`). The same happens once the workout is done; the person uses the mobile app to stop tracking the workout session (`STOP_TRIP`). The interaction diagram for the devices for this example is shown in Figure 1.

The state of the application in this example, will be defined by a vector $<HR, SYNC>$, where $HR$ is the heart rate sensor that can have values 0 (deactivated) or 1 (activated), and `SYNC` is the data synchronization activity with values 0 (stopped), 1 (started). This is shown in figure 2. Initially, both $HR$ and $SYNC$ are stopped and the application has state $<0, 0>$. After receiving the `START_TRIP` message the application goes to state $<1, 1>$. Then, after receiving `STOP_TRIP`, the state goes to $<0, 0>$.

In general, if there are $N$ sensors in the device, then the state of an application can be represented by a tuple $<S_1, S_2, .., S_N, SYNC>$. Even though wearable devices can have in excess of 20 sensors leading to many possible states in theory, we found that, in practice, a given application uses only a few sensors. We, therefore, restrict the sensors in the state definition to those used in the app. Applications also often activate sensors in groups, e.g., an application with 3 sensors can go from state $<0, 0, 0, 0>$ to $<1, 1, 1, 1>$ without traversing through other combinations of the sensors leading to further reduction of the state space. We found that this relatively coarse definition of states reduces the state-space size but is sufficient for our state-aware fuzzing to lead to more fault activations.

Definition 1. *State Transitions. A wearable application changes its state from $ST_1$ to $ST_2$ in response to* events*, where* events *can either be UI events, Intent messages, or synchronization messages.*

Example of UI event is tapping on a button, whereas, example of Intent message is launching the dialer app by sending $ACTION\_DIAL$. Example of synchronization message is presented in Figure 1.

## 3.2 Ewok Components

One of the design goals for Ewok was to make it less intrusive and keep the wearable device and the wearable apps (our fuzz target) unmodified. The overall architecture of our testing tool and the interaction between each of the components is shown in Figure 3. Since Ewok targets both communication within the wearable device (inter-process communication via Intents) and across devices (mobile and wearable), its components are deployed across the devices.

**Orchestrator.** The orchestrator is the core of Ewok which determines what actions will be performed next. Given a target application and its corresponding state model, the orchestrator can run the application in one of two modes–replay mode and fuzz mode. In the replay mode, the orchestrators guides an application to a target state, while in the fuzz mode, fuzzed messages are injected into the application. The orchestrator targets every state in the state model (of an application) sequentially. For example, it first fuzzes the application in state $ST_0$, then guides the application to state $ST_1$ by replaying normal traces from state $ST_0$. Next, fuzzing is done in state $ST_1$ and so on. The replay and fuzz cycle happens for each state in the state model. In theory, states with higher concurrency (e.g. more sensor activation, and more synchronization) will have higher failure rates [34]. We validate this hypothesis, by fuzzing every state in the model. Once a set of vulnerable states are discovered, these can be targeted in other apps with similar state model.

**Replay Module.** The replay module is responsible for guiding the target application to a given state. Based on the state model of the application and its normal traces, it first determines the sequence of UI events needed to drive the application to a target state. The ordered set of UI events is injected into the mobile device using ADB (Android Debug Bridge) [16]. In case of a crash on the wearable application, the replay module steers the application to last valid state before the failure.

**Fuzz Executor.** Ewok supports two different types of fuzzing strategies depending on the state of the application. While Intent Injection targets inter-component communication within the same device (wearable), Communication Fuzzing targets messages across devices (mobile and wearable). Depending on whether a state has ongoing data synchronization ($SYNC$), either Communication Fuzzing or Intent Injection is applied. *A. Intent Injection.* Random generation of explicit intents targeting *Activities* and *Services* within an application. We modify the Action and Data fields of intents based on three pre-defined fuzz strategies: *semi-valid* (use combination of known Action and Data values), *random* (use random strings as Action or Data) and *null or empty* (keep the fields blank). Intent injection is applied in all states where *no* data synchronization happens between mobile and wearable (i.e. $SYNC = 0$). For instance, using a semi-valid strategy, the fuzzer will generate an `Intent{cmp=WearAct, act=action.RUN}`, when the application is expecting: `Intent{cmp=WearAct, act= fitness.TRACK}`.

*B. Communication Fuzzing.* The data that is being synchronized from mobile to wearable device is fuzzed. In this case, we fuzz the communication messages based on two strategies: *random* (use random data value) or *null or empty* (use *null* value). Communication fuzzing is done in states where $SYNC = 1$. For example, for a message like `[/getOffDismissed, SomeMessage]`, the fuzzer replaces the message with `[/getOffDismissed,`

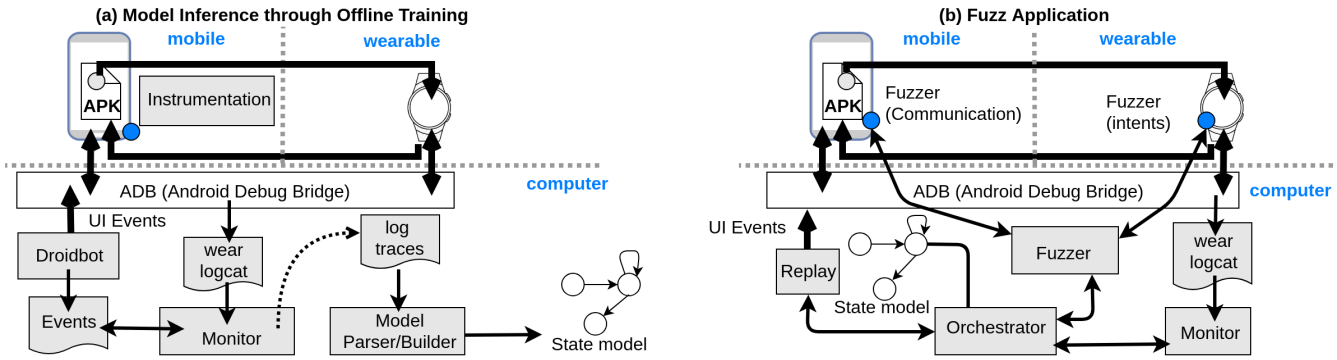


**Figure 3: Ewok architecture.** The diagram shows the location for various components and their interaction in each of the two stages: (a) The tools collect information from the log traces to build a model that represents the application; (b) The model is used to guide fuzz campaigns targeting the wearable application.

`null]` using an null or empty strategy. In either of these campaigns, if the application state changes while fuzzing, the control goes back to the *orchestrator* to decide the next fuzzing strategy to execute.

**Instrumentation Module.** The instrumentation module is crucial for fuzzing inter-device communication messages between the wearable and the mobile. Since we did not want to alter the functionality or behavior of the target application or device, we needed a lightweight method to intercept messages between devices and to modify them. We achieve this by registering a callback function to the communication methods provided by the Google Play Service library in the Data Layer API. More details is presented in the Implementation section.

**Monitor**. The monitor keeps track of current state of the wearable app during the fuzz testing.

## 4. IMPLEMENTATION

This section describes major aspects of the implementation. We implemented our testing tool using Java, Python, and Javascript. Ewok does not require access to source code of the target application and can work with binaries. This is of course critical for adoption of our tool and is different from many tools in this space, such as ACTEve [3], JPF-Android [35]. All our instrumentation and fuzzing are done without any modification to the application logic itself. In order to fuzz inter-device communication messages we alter the message at source (i.e. on the mobile side). This was achieved by dynamically instrumenting the mobile app and registering a callback to its Data Layer API calls. The instrumentation module was implemented using the Frida [4] framework which allows dynamic binary injection on applications already loaded in memory.

The fact that Ewok is designed to fuzz applications without access to the source code posed a challenge to infer the *state model*. The relevant events to our approach, interaction between devices and hardware sensor activation, are observed using the `adb logcat`, `dumpsys`, and the communications traces collected from the callback functions (instrumentation). Using this information, Ewok builds the state model for the target application. The communication between a wearable and mobile device is supported by Google Play services framework relying on the `Wearable Data Layer API`. We collected the log traces of inter-device communication by enabling logging in Android's `WearableListenerService`.

In contrast, in the case of the hardware sensors, sensor events are already present on the Android log trace. This output corresponds to the Android HAL (Hardware Abstraction Layer) [5] implemented for the devices; for example, when an application registers or unregisters a listener for a specific sensor. We complement the information collected from the log traces with the status of the sensors from the `dumpsys` service, available from the `adb` (Fig. 3).

## 5. EXPERIMENTS

For the experiments, we used a mobile phone (Google Nexus 6P, Android 7.1, released Apr 2017) paired with a smartwatch (Moto 360) with Android Wear 2.0 (released in 2017) installed. During the experiments, the mobile was connected to a computer via USB to collect the log traces using `adb`, while the smartwatch was connected to `adb` over Bluetooth. We analyzed the log traces to generate the results shown here.

### 5.1 Setup

We evaluated our fuzz tool on 15 wearable apps, downloaded from the Google Play Store and installed in both devices. We cover the four most popular categories of apps in the wearable space—in order, health and fitness, maps and navigation, shopping, and a rather non-descript category called personalization. All the apps selected have a companion app paired in the mobile device. We chose mature and popular apps, based on the number of downloads (at least 1M), and based on recognized popularity rankings [7, 19, 22]. Table 1 shows the detailed information on the selected apps. Although

**Table 1: Overview of selected apps for the evaluation.** All the apps were downloaded from Google Play store between Oct 2018 and Jan 2019. Sensors: Ac: Accelerometer, Gy: Gyroscope, Hr: Heart Rate, Ob: Low Latency Off Body, Up: User Profile, Uf: User Stride Factor. Categories: H&F: Health & Fitness, Maps: Maps & Navigation, Shop: Shopping, Wthr.: Weather, Pers.: Personalization

| App Name | Ver. | Cat. | Dwnld. | #Act. | #Srv. | Sensors |
|---|---|---|---|---|---|---|
| Google Fit | 2.03 | H&F | 10M+ | 61 | 17 | Ac,Gy,Hr,Ob |
| Strava | 10.0 | H&F | 10M+ | 10 | 8 | Ac, Hr |
| Citymapper | 7.17 | Map | 5M+ | 11 | 6 | |
| Runkeeper | 3.1 | H&F | 10M+ | 11 | 4 | |
| Runtastic | 4.0 | H&F | 1M+ | 13 | 9 | Hr |
| Seven Minute Workout | 7.1 | H&F | 1M+ | 6 | 4 | Hr |
| Muzei Live Wallpaper | 2.5 | Pers. | 1M+ | 5 | 12 | |
| Heart Rate Plus | 2.5 | H&F | 1M+ | 2 | 4 | Hr |
| Water Drink Reminder | 4.28 | H&F | 10M+ | 5 | 5 | |
| Bring! | 3.21 | Shop | 1M+ | 16 | 16 | |
| Nike Run Club | 1.1 | H&F | 10M+ | 6 | 7 | Ac,Hr |
| MyRadar Weather | 7.4 | Wthr. | 10M+ | 11 | 4 | |
| Keep Trainer | 1.2 | H&F | 5M+ | 4 | 2 | |
| Google Maps | 10.4 | Map | 5B+ | 28 | 18 | |
| Moto Body | 1.0. | H&F | 1M+ | 21 | 7 | Hr,Up,Uf |
| **Total** | | | | **210** | **123** | |

the number of apps in our sample set is small (15), combined they had a total of 210 activities and 123 services, therefore, we consider this as a representative subset of wearable application components. These are the apps for which we did the detailed study, including root cause analysis of failures, and complemented this with a shallower study of a larger number of apps (100).

## 5.2 State-aware Injection Campaigns

One of the primary objectives of Ewok is to evaluate the effectiveness of a state-aware fuzzing strategy in comparison with state-agnostic fuzzing. To this end, we ran our experiments against the apps in Table 1 with fuzzing strategies described in Sec. 3.2. Based on the state of an application, our fuzzer either alters intra-device communication messages (Intents) or inter-device communication messages (data synchronization messages). We hypothesize that altering global state of the device (how many and which sensors are activated) can lead to different failure manifestations. Coverage may be increased by activating or deactivating arbitrary sensors from within the app. However, doing so would require alteration of the target application, which is outside of our usage model. Therefore, we decided to stress the device, and thus the apps, by activating sensors through an external application (we call it the "Manipulator app"). The Manipulator app either activates the same set of sensors as those in an app (thereby causing contention for those sensors) or it activates the complementary set of sensors (increasing load on the `SensorManager`, the base class that lets apps access a device's sensors). Thus, we ran Ewok under three scenarios as shown in Table 2.

**Table 2: List of experiments.**

| Experiment | Device State |
|---|---|
| I | Not modified |
| II | Activate complementary sensors |
| III | Activate same sensors as in the target app |

Some of the apps in our sample set did not use any sensor (see Table 1 with the last column blank), therefore, Expt III is not applicable to these apps. Due to this, we split the results into two categories: (a) apps that use sensors for which all three experiments are run and (b) apps that do not use sensors for which only experiments I and II are run. We found that despite not using any sensors, these apps still showed larger number of crashes and reboots when Expt II was applied.

We compared the results of Ewok against state agnostic testing tools, QGJ [9], Android Monkey tool [6] and Droibot [24]. QGJ, a black box fuzz testing tool, injects malformed intents to wearable apps based on four campaigns which vary the intent's input (e.g., *Action*, *Data* or *Extras*). On the other hand, Monkey generates pseudo-random UI events and injects them to the device or emulator. We assigned an equal probability to each of the possible events in Monkey. Finally, Droidbot offers similar capabilities as Monkey, but the tool tries to maximize the coverage of the states in the apps by using a model based on the GUI of the application.

## 5.3 Failure Categories

From our experiments, we found failures to have three types of manifestations. These are all user-visible failures leading to app crash or freezing or more worryingly reboot of the entire system. **ANR:** The system generates a log entry containing the message `ANR` (Application Not Responding). This condition is triggered when

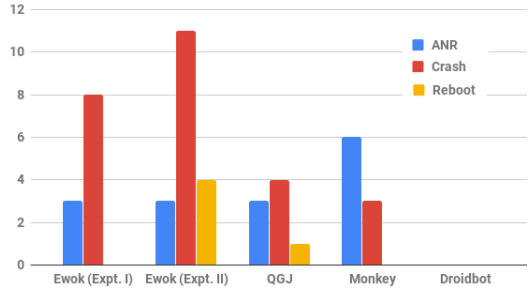


**Figure 4: Distribution of failure manifestation for all the Tools.** This shows that state-aware fuzzing led to more crashes and reboots than state-agnostic tools like QGJ and Monkey.

the system is not responding because it is waiting for a process to finish. **Crashes:** The system generates a log entry with a message `FATAL EXCEPTION`. This corresponds to an application crash. **Catastrophic Failures:** The system generates a log entry with the message `android.os.DeadSystemException`. This is the most severe of the failures since it triggers a soft reboot of the Android runtime. These manifestations are equivalent to the Catastrophic, Restart, and Abort failures proposed by [23] in the classification used to evaluate the robustness of server OSes. Another interesting class of errors, silent data corruption, is kept outside the scope as it needs expert users writing specific data validity checks. We also capture the stack trace following a failure message to identify unique failures (or bugs). Hence, if the failure is triggered multiple times by the same cause (i.e., it has the same stack trace), we only count that failure once.

## 5.4 State-Aware Injection Results

Figure 4 and Tables 3-4 show the results obtained from our experiments. Since, Expt III is not applicable to the apps that do not use any sensors, these results are shown separately in Table 4. The numbers shown for Expt I and Expt II in Fig. 4 are aggregate numbers from Tables 3, 4. We omit Expt III from Fig. 4 since it has fewer apps than other experiments and cannot be compared directly. In general, we found that our state-aware fuzzing with Ewok generated more crashes and reboots than state-agnostic fuzzers like QGJ and Monkey. No crash was observed with DroidBot, which only activates various UI components. Monkey generated more ANRs compared to Ewok, since it sends events at a higher rate than Ewok. Ewok pauses injections after a set and invokes the Garbage Collector to prevent overloading the system. From Table 4, we see communication fuzzing is effective when all sensors are activated showing that the apps and device are in a vulnerable state when synchronization and sensor activation are occurring concurrently.

We also found differences in failure manifestations across Experiments I, II and III, which supports our hypothesis that changes in the device state can trigger new bugs on the wearable app. More specifically, we found that number of crashes in Expt II is either same as those in Expt I (for apps that use sensors) or greater (for apps that do not use sensors). Similarly, number of reboots is higher in Expt II and III compared to Expt I. This indicates that sensor activation (even through an external app) has a negative effect on overall reliability of the system. A state-aware fuzzer like ours can automatically guide a wearable application to a vulnerable state and then fuzz the state, thus increasing the rate at which bugs can be discovered. As part of responsible disclosure, we have reported the details of app crashes and system reboots to the vendors.

**Table 3: Failure manifestations for apps that use sensors.** This clearly indicates that state-aware approach can trigger more failures than state-agnostic approaches like QGJ or Monkey. For Ewok all the failures correspond to the Intent fuzzing strategy.

| State | #ANR | #Crashes | #Reboots |
|---|---|---|---|
| Ewok (Expt. I) | 3 | 4 | 0 |
| Ewok (Expt. II) | 2 | 4 | 2 |
| Ewok (Expt. III) | 4 | 5 | 2 |
| QGJ | 3 | 4 | 1 |
| Monkey | 5 | 2 | 0 |

**Table 4: Failure manifestations for apps that do not use sensors.** This also indicates that altering device state can lead to more ANR, crashes and reboots. For Ewok, the values are presented in parenthesis as (Intent fuzzing, communication fuzzing).

| State | #ANR | #Crashes | #Reboots |
|---|---|---|---|
| Ewok (Expt. I) | 0 (0,0) | 4 (0,4) | 0 (0,0) |
| Ewok (Expt. II) | 1 (1,0) | 7 (3,4) | 2 (2,0) |
| QGJ | 0 | 0 | 0 |
| Monkey | 1 | 1 | 0 |

Fig. 5 shows the distribution of exception types across the experiments. Unlike in Fig. 4 where only unique failures are counted, here we count all such manifestations. For each experimental run where there is a failure, we add a count of 1 to the exception that was seen last in the log before the failure. It can be seen that `NullPointerException` dominates most of the failures. This exception is generated by the Android runtime when an application does not receive an expected input, however, the user application does not catch this exception leading to crashes. Previous work [9, 29] has shown that such input validation bugs have been a common category in Android apps over the years. Our results indicate that similar situation persists in current generation of wearable apps.

## 5.5 State-agnostic Injections

Next, we evaluate the Wear OS ecosystem with state-agnostic injections. The only previous work was done by Barsallo *et al.* [9], who conducted a study on 46 wearable apps using a state-agnostic fuzzing tool QGJ. For our study, we selected 40 built-in and 60 popular third-party apps on the Google Play store [18]. The selection includes applications from different categories, such as Health & Fitness, Tools, Maps & Navigation, and Productivity. To enable the comparison with QGJ, we stripped out the novel state-aware functionality of Ewok and used it to fuzz inputs in a state-agnostic manner, and injected over 600K Intents to Service and Activities components (a total of 1,530 components). Our evaluation differs from [9] in that we included most of the built-in apps and extended our focus beyond only Health & Fitness applications. We found that, only 1.9% of the injections produced a failure. Among the failures (refer Figure 6), Crashes dominated the list

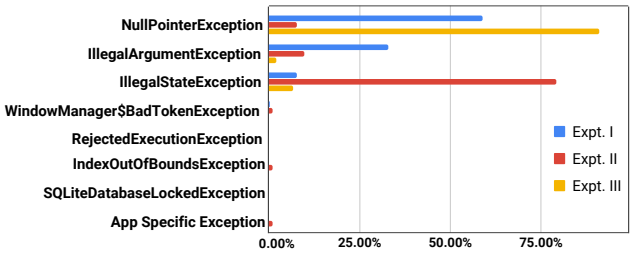


**Figure 5: Distribution of Exception Types across runs that resulted in a failure.** The distribution of exception types appears vastly different in Expt. II because the app that generated bulk of the NullPointerExceptions in Expt. I and III (Moto Body), rebooted the wearable device while Expt. II was running, causing our fuzzer to ignore rest of its components.

(98.6%), followed by ANR (1.3%) and Reboots (0.1%).

To have a quantitative comparison with the injections of QGJ, we also computed our failure manifestations as a percentage of the components that are affected. Table 5 shows the complete breakdown of the manifestations per application component. We follow a similar approach as [9], and classified every component according to any of the possible manifestations. For those components with multiple manifestations, we chose the most severe one. We found that over 94% of the components did not experience any failure, the remaining 6% of the components expressed at least one failure manifestation. In contrast, the previous study reported that 13.8% of the components were affected by a failure manifestation. This difference can be explained by the fact that our evaluation includes a larger sample of apps (not just Health & Fitness apps) and the overall set tends to have simpler apps than Health & Fitness, which often have to deal with multiple sensors and complex synchronization with mobile devices.

**Table 5: Distribution of failure manifestation for state-agnostic injections.** Percentage is with respect to all the components targeted through the injections.

| Manifestation | % |
|---|---|
| No Effect | 94.71 % |
| ANR | 1.18 % |
| Crash | 3.53 % |
| Reboots | 0.59 % |

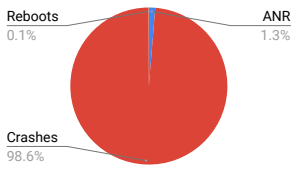


**Figure 6: Distribution of failure manifestation for state-agnostic injections for 100 apps.**

## 5.6 Root Cause Analysis

In this section, we present various interesting failures observed during our experiments. These failures highlight some nuances about the Android Wear ecosystem and provide motivation for future research.

**Android-Wear OS Code Transfer.** Among the crashes observed, we found that every time our fuzzer injected a `KEYCODE_SEARCH` key, the wearable app crashed. We decided to do further experiments with other key events defined in Android [17], and no other event triggered a similar vulnerability. On Android, the `KEYCODE_SEARCH` event is used to launch a text search inside the application (apps can also silently ignore the event if it has not implemented this feature). However, in the case of Wear OS, when the event is injected, the apps crashed with an `IllegalStateException`. Digging deeper we found that Wear OS, when inheriting code from Android, removed the `SearchManager` service class, assuming probably that text searches are not suited for the form factor of wearables. This kind of behavior can be seized on by a malicious application (with system-level privileges however) to crash other applications. This is a widespread problem—of the set of 100 applications we used for the state-agnostic evaluation, just one application (Google Play Store) was resistant to this kind of attack. The root problem can be handled at the Wear OS level by simply discarding the search event in the base Activities class (`WearableActivity`) instead of trying to launch `SearchManager`.

**Error Propagation.** Ewok found vulnerabilities related to ongoing synchronization between mobile and wearable, where 43% of the communication injections led to failure. It is noteworthy that one of these failures, on `Citymapper` app, even propagated back to the mobile device, crashing not only the wearable app but also its mobile counterpart. The crash of the wearable application is due to `IllegalStateException`. This triggers a communication from the wearable to mobile which passes the `Exception` object, however, the message sent by the wearable triggers a `NullPointerException` in the mobile app when it tries to invoke a method `getMessage`. This exception is caused by a mismatch of the serialized object sent by the wearable and that expected by the mobile. The wearable sends the exception wrapped as a `GSON` object, while the mobile device expects a `Throwable` object.

**Catastrophic Failures.** We decided to do further analysis of the root cause of some of the catastrophic failures, the system reboots. All of these failures are characterized by the exception `DeadSystemException` in the adb logs. It is to be noted that Ewok can trigger system reboot without any root privilege. We determined that the main reason for system reboots in Wear OS is starvation of resources. Previously, [20] found vulnerabilities in the concurrency control mechanism of the System Server (SS) in Android. The authors showed that due to coarse-grained locking in Android SS, several Android APIs are vulnerable to DoS attacks, where a user-level application can keep the SS busy for a very long time and thereby starving other applications. If the starvation exceeds a pre-set deadline, a watchdog

process kills the Android SS and restarts the system. We see similar freezing behavior by activating multiple sensors while fuzzing. It may be noted that [20] did not detect any vulnerabilities in the `SensorManager`, but our experiments indicate that higher sensor use leads to more reboots pointing to concurrency handling problem in this foundational component.

# 6. LESSONS AND THREATS

**Input Handling.** Input handling of Intents is still a major cause for crashes in the Wear OS—although frequency of `NullPointerException` has reduced over the years (and it still remains the single most common class), the frequency of `IllegalArgumentException` has increased. We found that input handling in built-in applications is often misunderstood by app developers. A common pattern is: validation is done by a system service class and that raises an exception, the app using the class does not handle the exception and crashes. This can be handled better in IDEs such as Android studio during app development. If a system service (e.g., Google Fit in our case) throws an exception, the tools should check absence of exception handling codes at compile time and throw an error.

**Error Propagation.** Another type of input validation–those related to error propagation across devices–can be handled better by additional knowledge of the data type being shared between devices. For instance, WSDL is used to define message exchange among web services. A similar approach can be followed, without exposing the details in the APK, to define the data types being sent by the endpoints in the manifest files. Then, the IDE tools can use this definition to raise error during compilation when a type mismatch is detected.

**Watchdog and Resource Starvation.** Our results show that it is possible to trigger catastrophic failure on Wear OS without any root privilege. This manifestation was always preceded by a resources starvation of the system due to processes in the user level accessing hardware sensors. Currently, the watchdog process elects to kill a system level thread (such as the `SensorService`) to free up resources. A better design is needed, in which the OS avoids reaching such starvation, either by electing to kill abusive *user-level* processes or by declining registration of additional sensors if the system finds itself overloaded.

**Threats to Validity.** The experiments presented in this paper have few shortcomings which may introduce biases. First, our study is based on results on a single wearable device and a single mobile phone, which can be impacted by vendor-specific customizations. Second, our inter-device communication fuzzing is currently one-directional–we only fuzz messages going from the mobile to the wearable. This is primarily to keep the wearable device unaltered and because the injection tool Frida only exists for Android. Third, even though we selected popular apps from different categories, our sample size of 15 (for state-aware injections) or even 100 (for state-agnostic injections) can be considered to be small.

# 7. RELATED WORK

**Stateful Fuzzers.** Most of the previous work on stateful fuzzing have been focused on communication protocols, such as SIP or FTP [8,13–15]. A common approach here is to infer a finite state machine by analyzing the network traffic. Then, this model is then used to guide the black-box fuzz testing. Our work differes from these fuzzing tools in that our state model captures application state as opposed to protocol state.

**Android.** Since its release in 2008, there have been numerous studies on the reliability of Android OS. These works vary from random fuzzing testing tools based on UI events and system events [28, 36], fuzzers focused on Android IPC [10, 29, 32], model-based testing tools based on static analysis [1] or specialized approach based on concolic testing [3]. Previous works based on model representations of the apps, focused on GUI navigation [2, 12, 24, 31, 33, 38], rather than a stateful approach of communication and sensors as in Ewok.

**Wear OS.** Research on Wear OS focus on the performance and the reliability of the OS itself [25–27]. However, recent studies have addressed the need for testing tools for the Wear OS ecosystem. [9, 37]. Barsallo *et al.* [9] performed a robustness study to analyze the reliability of the Wear OS ecosystem. They proposed QGJ, a stateless fuzzer that relies on fault intent injection. To the best of our knowledge, Ewok is the first tool that fuzz test Wear OS using a stateful approach.

# 8. CONCLUSION

In this paper we presented the design and implementation of Ewok, a state-aware fuzzing tool for wearable apps. Ewok can automatically build a state model for an app from its logs and steer the app to specific states by replaying the traces. It then launches state-aware fuzzing on the wearable app targeting both inter-device and intra-device communication. We found that state-aware fuzzing can lead to more crashes of the apps and reboots of the device compared to stateless fuzzing. We also found that it is possible to predictably reboot a wearable device from a user app by targeting specific states, namely, states with concurrent sensor activity coupled with synchronization between the mobile and the wearable. We draw lessons for improving the wearable ecosystem—single code point handling of exceptions, type checking of communication between mobile and wearable devices, and diagnosing and terminating *user-level* components that are causing resource starvation for sensor resources. Our future work will focus on automatically and efficiently guiding an app to a vulnerable state thus speeding up the vulnerability discovery through stateful fuzzing.

## 9. REFERENCES


[1] Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., and Memon, A. M. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (2012), ACM, pp. 258–261.

[2] Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., and Memon, A. M. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2012), ASE 2012, ACM, pp. 258–261.

[3] Anand, S., Naik, M., Harrold, M. J., and Yang, H. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (2012), ACM, p. 59.

[4] Andre, O. Frida, 2018.

[5] Android. Sensor hal interface, 2017.

[6] Android. UI/Application Exerciser Monkey, 2017.

[7] AndroidRank. Free Android Market Data, History, Ranking 2011-2019, 2019.

[8] Banks, G., Cova, M., Felmetsger, V., Almeroth, K., Kemmerer, R., and Vigna, G. Snooze: toward a stateful network protocol fuzzer. In *International Conference on Information Security* (2006), Springer, pp. 343–358.

[9] Barsallo Yi, E., Maji, A. K., and Bagchi, S. How reliable is my wearable: A fuzz testing-based study. In *DSN* (2018), pp. 410–417.

[10] Burns, J. Intent Fuzzer, 2012.

[11] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing inter-application communication in android. In *Mobisys* (2011), ACM, pp. 239–252.

[12] Choi, W., Necula, G., and Sen, K. Guided gui testing of android apps with minimal restart and approximate learning. In *Acm Sigplan Notices* (2013), vol. 48, ACM, pp. 623–640.

[13] Comparetti, P. M., Wondracek, G., Kruegel, C., and Kirda, E. Prospex: Protocol specification extraction. In *Security and Privacy, 2009 30th IEEE Symposium on* (2009), IEEE, pp. 110–125.

[14] De Ruiter, J., and Poll, E. Protocol state fuzzing of tls implementations. In *USENIX Security Symposium* (2015), pp. 193–206.

[15] Gascon, H., Wressnegger, C., Yamaguchi, F., Arp, D., and Rieck, K. Pulsar: stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems* (2015), Springer, pp. 330–347.

[16] Google. Android Debug Bridge, 2017.

[17] Google. Android Developers. KeyEvent, 2019.

[18] Google. Top Free in Wear OS by Google, 2019.

[19] Hindy, J. 10 best Android Wear apps (Wear OS apps), 2018.

[20] Huang, H., Zhu, S., Chen, K., and Liu, P. From system services freezing to system server shutdown in android: All you need is a loop in an app. In *ACM CCS* (2015), pp. 1236–1247.

[21] Iannillo, A. K., Natella, R., Cotroneo, D., and Nita-Rotaru, C. Chizpurfle: A gray-box android fuzzer for vendor service customizations. In *ISSRE* (2017).

[22] Johnson, L. Best Android Wear Apps: 13 essential free downloads for your smartwatch, 2018.

[23] Koopman, P., Sung, J., Dingman, C., Siewiorek, D., and Marz, T. Comparing operating systems using robustness benchmarks. In *Proceedings of SRDS'97: 16th IEEE Symposium on Reliable Distributed Systems* (1997), IEEE, pp. 72–79.

[24] Li, Y., Yang, Z., Guo, Y., and Chen, X. Droidbot: a lightweight ui-guided test input generator for android. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on* (2017), IEEE, pp. 23–26.

[25] Liu, R., Jiang, L., Jiang, N., and Lin, F. X. Anatomizing system activities on interactive wearable devices. In *APSys* (2015), pp. 1–7.

[26] Liu, R., and Lin, F. X. Understanding the characteristics of android wear os. In *Mobisys* (2016), pp. 151–164.

[27] Liu, X., Chen, T., Qian, F., Guo, Z., Lin, F. X., Wang, X., and Chen, K. Characterizing smartwatch usage in the wild. In *Mobisys* (2017), pp. 385–398.

[28] Machiry, A., Tahiliani, R., and Naik, M. Dynodroid: An input generation system for android apps. In *FSE* (2013), pp. 224–234.

[29] Maji, A. K., Arshad, F. A., Bagchi, S., and Rellermeyer, J. S. An empirical study of the robustness of inter-component communication in android. In *DSN* (2012), pp. 1–12.

[30] Maji, A. K., Hao, K., Sultana, S., and Bagchi, S. Characterizing failures in mobile oses: A case study with android and symbian. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on* (2010), IEEE, pp. 249–258.

[31] Mao, K., Harman, M., and Jia, Y. Sapienz: multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (2016), ACM, pp. 94–105.

[32] Sasnauskas, R., and Regehr, J. Intent fuzzer: crafting intents of death. In *WODA and PERTEA* (2014), pp. 1–5.

[33] Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., and Su, Z. Guided, stochastic model-based GUI testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), ACM, pp. 245–256.

[34] Sullivan, M., and Chillarege, R. Software defects and their impact on system availability-a study of field failures in operating systems. In *[1991] Digest of Papers. Fault-Tolerant Computing: The Twenty-First International Symposium* (June 1991), pp. 2–9.

[35] van der Merwe, H., van der Merwe, B., and Visser, W. Verifying android applications using java pathfinder. *ACM SIGSOFT Software Engineering Notes 37*, 6 (2012), 1–5.

[36] Ye, H., Cheng, S., Zhang, L., and Jiang, F. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia* (2013), ACM, p. 68.

[37] Zhang, H., and Rountev, A. Analysis and testing of notifications in android wear applications. In *ICSE* (2017), pp. 347–357.

[38] Zheng, H., Li, D., Liang, B., Zeng, X., Zheng, W., Deng, Y., Lam, W., Yang, W., and Xie, T. Automated test input generation for android: towards getting there in an industrial case. In *Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), 2017 IEEE/ACM 39th International Conference on* (2017), IEEE, pp. 253–262.